5-9-2006

A Comparative Study of State Emulation in Functional Programming Languages

William Brick
Trinity University

A Comparative Study of State Emulation in Functional Programming Languages

William Brick

A departmental thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for Graduation.

April 1, 2006

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

A Comparative Study of State Emulation in Functional Programming Languages

William Brick

Abstract

We examine the methods of emulating state in functional languages. In particular, we investigate the languages J, Scheme, and Haskell; the latter two being representative of the Lisp and ML families, respectively. We provide example code for state emulation in terms of object-orientation and compare this to object-oriented programming without use of state.

Acknowledgments

I'd like to thank my advisor, Dr. Howland, for his help with this project, as well as my thesis committee members Dr. Eggen and Dr. Holder.

A Comparative Study of State Emulation in Functional Programming Languages

William Brick

Contents

List of Tables

Chapter 1

Introduction

In this paper we examine methods of state emulation in J, Scheme, and Haskell, the motivation being that we might obtain examples of widely different techniques. Each of these languages takes a different approach to state emulation, and these three approaches are employed in a wide variety of languages. For example, J uses special locales to create an entire object system, whereas Scheme encapsulates a state change inside of lexical closures and Haskell uses a special monad referred to as a "state transform monad". Each of these techniques is examined in depth and used to implement a simplistic semaphore object. We also discuss a purely functional implementation of this object, evaluating the utility and efficiency of each implementation.

1.1 Motivation for Study

Before we begin, it is crucial to understand the necessity for such a study. We posit that stateful computation is rather important to the computer science community; most programmers would be upset to find this tool unavailable. Many algorithms are difficult or

impossible to code efficiently without some using some sort of state-change. For example, many graph theory algorithms and structure-modifying methods are difficult to reason about without using state, and even those that don't are often inelegant and difficult to read when coded in a non-stateful environment. Aside from this, state is necessary to perform any sort of input/output, and is therefore indispensable for any practical programming effort. Given all of this, it seems that no programming language is complete without some ability to perform stateful computations.

Why, then, should we not simply use imperative languages exclusively, abandoning the functional realm as one of theory and conjecture, of little practical value? To note just a few reasons, many concepts and ideas are elegantly expressed in functional programming and functional languages are often compact, readable, and easy to reason in. Features such as referential transparency, first class functions (allowing for currying), tacit programming, and non-strict evaluation are only a few of the desirable concepts that are difficult to attain or useless outside of functional programming. In fact, certain practices widely used in procedural programming are in fact much easier in functional programming; memoization, for example, becomes much simpler with guaranteed referential transparency and first-class functions. So, if it is impractical to lose the advantages of state but desirable to seek the advantages of pure functional programming, where does the solution lie? Several answers may be offered, but in this paper we explore the machinery used to emulate state and the potential advantages and disadvantages of such methods, including performance measurements designed to assess the practicality of such emulations.

1.2 Explanation of Methods of Evaluation

1.2.1 Object-Orientation

In this study we find it helpful to evaluate the methods offered by these languages in terms of several different criteria. Our reasons for choosing these criteria are based on the usefulness of the particular implementation to an average programmer. For that reason, one property we examine is how well our state emulation transfers to object-oriented programming. For most modern programmers, object orientation has become a staple of programming, and in fact has subtly influenced the way we often reason about programs. Whether one chooses to embrace the object model and use it continually or eschew it in favor of other techniques, object-oriented programming is inescapable. With this in mind, we examine each of our languages to see how well the state emulation matches an object-oriented model.

1.2.2 Ease of Use

Another criterion by which we judge our languages is the ease of use for the programmer. A language construct may be powerful, efficient, and elegant, but if it is difficult or complicated to implement, the programmer may nonetheless choose to use a different construct or even a different language to circumvent this obstacle. With this in mind, we examine the accessibility and usefulness of each of these constructs for the average programmer. This includes digressions on the amount of documentation readily available, the intuitiveness of the commands, and the complexity of the implementation. Those languages which are easy to understand, simple to code, and have plentiful and helpful documentation are obviously favored over ones that do not share these traits.

1.2.3 Efficiency

A third criterion by which we judge the languages and their constructs is the efficiency of the implementation. For this purpose we have run several speed tests of the stateful portions of J, Scheme, and Haskell. We have tested the creation and manipulation of stateful objects within these languages, and in this paper we examine the results of this creation and factor those into our evaluation of these languages.

Chapter 2

State Emulation in J

2.1 Explanation

The J programming language is unique in several respects. It combines many features found in few other languages, such as tacit programming and locales. With tacit programming, the programmer can code by combining functions with no specific reference to arguments. The classic example of this is the averaging function `+/%#`, which combines the functions `+/` (addition insert), `%` (divide), and `#` (tally) in a way to average a list of numbers [2]. This is possible because the composition rule for a function of the form `(f g h) x` is `(f x) g (h x)`, where `f` and `h` are monads and `g` is a dyad. Thus, under this system the J interpreter parses the function`+\%#` as "apply addition insert to an argument, apply tally to the same argument, and apply divide to the first result then the second result. In J, top level reassignments are permitted, but these reassignments are unlike reassignments in imperative programming. Under this scheme reassignments are simply assigning a name to a different value rather than assigning a new value to the name. Thus names are not tied to memory cells but are higher-level abstractions. Also, in J only toplevel reassignments

are allowed. J manages to emulate state safely through use of "locales", specifically the z locale [3]. A locale is a set of names visible only within the locale. For example, the function `fact_a_` (the `_a_` indicates that it's in locale a) will automatically run on data within locale a. Thus, the line `fact_a_ num` will return the value of the function `fact_a_` run on `num_a_`, if `num_a_` exists. In this locale system, the z locale is a special locale; it is a parent locale, or one to which every other locale refers. If a name is referred to in a locale, the interpreter first looks for it within the current locale, and if it does not find it there it looks in the z locale. However, if a name in the z locale is referenced from within some other locale, the interpreter behaves as though it was referenced from within the original locale rather than the z locale. So if `num_a_` does not exist and `num_z_` does, then `fact_a_ num` will return the value of the function `fact_a_` run on `num_z_`. Using this system, the programmer can create locale-specific state-based objects out of classes defined in the z locale, effectively transforming the locale into an object. Using a global `make` command defined in the z locale, the programmer can instantiate an object into any locale and safely alter the state of top level names. If this system was not in place, any name referred to by a function would be visible to any other function; however, within a locale the user can safely alter the value of a name without the repercussions associated with state change in a global environment.

2.2 Code

In the following code, we see the top level name `data` being set to zero as a default value. The code reassigns this name in several places, but all of the code shown below is instantiated into a specific locale, and the name `data` is not in the scope of any other locale.

```
make_z_ =: 0 !: 0 @ <
match =. -:
```

```
data =. 0

semaphore =. monad define
y.
if. y. match 'type'
    do. 'semaphore'
  elseif. y. match 'get'
    do. data
  elseif. y. match 'increment'
    do. data =: data + 1
  elseif. y. match 'decrement'
    do. data =: data - 1
  elseif. 1
    do. 'invalid method error'
end.
)
```

Chapter 3

State Emulation in Scheme

3.1 Explanation

The programming language Scheme [6] inherited some non-functional characteristics from its predecessor, LISP . One of these was the `set!` function, which is used to change the value associated with a name. This is a native construct in Scheme which was set in place to emulate state by the creators of the language. In the original implementation of LISP this could cause problems, because it was dynamically scoped, causing problems with globally scoped names. Thus, if one function uses a `set!` to change to the value of a variable, any other function can access that variable, and thus the behavior of the interacting functions may be awkward. For example, if a function `invert` changes the sign on an integer, the programmer would have to make sure that no other part of the code attempts to take the square root of that integer, since any other function could access or even change its value after `invert` changed the sign of the integer. In Scheme, an elegant solution to this problem was devised that relied on lexical scoping. With lexical scoping, names can only be referenced in the environment they are bound to. So, for example, depending on the input

given, our `invert` function could change the sign of an integer that only it can access, or return the value of that integer. These local bindings are known as syntactic closures [1]; they are usually achieved using `lambda` or `let` expressions. Binding a variable to one of these expressions ensures that it cannot be referred to outside of the expression, and thus creates a lexical closure in which the programmer can effectively use state change.

3.2 Code

The following example illustrates a simple semaphore created with a lexical closure. Note that the value of the name "cmd" will be released after each function call, but the value of the name "val" is retained between calls.

```
(define semaphore
  (let ((val 0))
    (lambda (cmd)
      (cond
        ((equal? cmd "get") val)
        ((equal? cmd "up")
         (set! val (+ val 1)))
        ((equal? cmd "down")
         (set! val (- val 1)))
        (else "Invalid Command")))))
```

With this object, when we make the following calls:

```
(define semx semaphore)
(semx "up")
```

```
(semx "up")
(semx "up")
(semx "get")
(semx "down")
(semx "get")
```

the output is

```
3
2
```

which indicates that the state of `semx` changes during execution.

Chapter 4

State Emulation in Haskell

4.1 Explanation

State emulation in Haskell is the most complicated method of the three, as it is based on mathematical category theory. The basic concept behind state emulation in Haskell is that the programmer may choose to simulate assignment to a mutable variable by passing a value representing the current state. Before we delve into this, however, we discuss the supporting concepts of state transformers and stateful monads.

4.1.1 The Haskell State Transformer

The state transformer in Haskell is essentially a purely functional account of the concept of a stateful variable, with type `(ST s a)` [4]. The computation transforms a mutable variable by taking as its input the old state and returning a new state. The simplest example is the state transforming implementation of the identity map, which returns the same state that it received as input; the type of such a transformer would be `ID :: a -> ST s a` [7]. Given this, it is simple to construct functions to read, write, and create mutable objects

and to create a new variable, For example, one would simply make a state transformer that takes a value and returns a mutable state object with this value. Other such transformers are relatively straightforward to implement, and we can even sequentially compose state transformers using certain tools such as `thenST`, which has type

```
thenST :: ST s a -> (a -> ST s b) -> ST s b
```

Unfortunately, we need a way to implement these transformers within the larger context of a purely functional program. This problem is cleverly solved using encapsulation.

4.1.2 RunST and Encapsulation

As a method of implementation, the designers of Haskell wrote the `runST` function. This function essentially creates a mini-environment in which state is manipulated as described above. When the user calls `runST`, the Haskell compiler creates a new thread in which a valueless state is initialized. It then runs through its computation and returns the value of the final state, discarding the state in the process. In *State In Haskell* [4], Launchbury and Jones provide a proof that multiple calls to `runST` cannot create a situation where threads reference state variables that belong to another thread; this proof is rather involved and depends on the parametricity of `runST`.

4.1.3 Implementation

The same reasoning behind the implementation of `runST` is used in the Haskell state monad, only on a higher level. In fact, much of the difference is syntax that helps the programmer manipulate the state transformer. The basic operations of the state monad are embodied in the `set` and `get` combinators. These manipulations are what one would expect from the names, as they are implementations of the reading and writing transformations we

considered earlier. There is one fundamental combinator used when executing a stateful program, and it is `runState`. As the name suggests, it is functionally similar to `runST`, and it returns a tuple of the result of the computation and the final value of the state object. Two other combinators, `evalState` and `execState`, return only the result of the computation and only the final value of the object, respectively. The point of `execState` is to examine the effects of the computation, since the result of the computation may not be directly tied to the final state of the state object.

4.2 Code

The following example shows a brief implementation of the semaphore in Haskell.

```
semaUp :: State Int ()
semaUp = do {x <- get; put (x+1)}
semaDn :: State Int ()
semaDn = do {x <- get; put (x-1)}
semaMk :: Int -> State Int ()
semaMk n = put n
```

With this object, it is important to note that we cannot make toplevel state changes. However, examine the following function:

```
semaManip :: State Int ()
semaManip = do
    semaMk 0
    semaUp
    semaDn
```

```
semaUp
```

If we run this state computation function with `runState semaManip 0`, we get `((),1)` as output. This is expected, since we have not technically specified any output. The output of the function is () and the resulting state is 1.

Chapter 5

Evaluation

In this chapter we review the evaluations and offer a brief comparison between languages.

5.1 Individual Evaluation

5.1.1 J Evaluation

In this section we evaluate the state emulation implementation system in the J programming language by the criteria previously mentioned. We start by examining the usefulness of the system in terms of object systems.

Object-Oriented Programming

The method of state emulation in the J programming language is ideal for creating an object system. The locale system provides a clean interface with the state object while still sufficiently encapsulating the stateful computation. In fact, the small integer implementation shown above is an object model, albeit a small one. Using this system, the programmer can create rather complex hierarchies of objects within certain locales that are encapsulated

entirely within the environment in which they are instantiated but are available to any part of the global environment. In terms of programming object systems, J is a valuable language.

Ease of Use

Of the three criteria, J is most deficient in terms of ease of use. The author found that writing the code for the J state emulation is the most difficult of the three languages. The J syntax is difficult to master, particularly for a programmer unfamiliar with the functional paradigm. This difficult syntax extends to the locale system, which can also be trying for an inexperienced programmer due to the fact that the symbols of the J language have no link to a common meaning. This is in contrast to Scheme or Haskell where, for example, the construct "eq?" can be remembered as a dyad that returns true when its arguments are equivalent and false otherwise. Additionally, the concept of locales, while not a particularly complex one, is rare in other programming languages. While many programming languages have similar features, few include the same concept of locales. Aside from this, while J documentation is readily available online, often the community support is not as strong as in other languages. The J Software website has a free online manual for the programming language, but should the programmer require more examples or a different explanation, it is often difficult to find further resources.

Speed

In terms of speed, J is an excellent language. The tests were run on the J Console (version 5.04). Tables 5.1 and 5.2 show the results of the tests, which yielded relatively good test times. Unfortunately, this was taken from within the J console, so in many applications the time taken to start and end a session within this console must be added to the total.

	J Console
Minimum Allocation	2.97×10^8
Maximum Allocation	6.02×10^8
Average Allocation	2.69×10^8
Minimum Manipulation	2.69×10^8
Maximum Manipulation	6.08×10^8
Average Manipulation	5.61×10^8

Table 5.1: J UNIX tests (in operations/sec)

	J Console
Minimum Allocation	1.89×10^8
Maximum Allocation	6.59×10^8
Average Allocation	5.77×10^8
Minimum Manipulation	1.90×10^8
Maximum Manipulation	6.61×10^8
Average Manipulation	5.58×10^8

Table 5.2: J Windows tests (in operations/sec)

However, despite the inherent slowness of any interpreted language, the J state emulation construct offers excellent performance under speed testing.

5.1.2 Scheme Evaluation

In this section we examine Scheme based on the same criteria outlined earlier. The first of these criteria is the ability to create an object system.

Object-Oriented Programming

Objects in Scheme are fairly straightforward. Like most features of the language, the lexical closures from which state models are derived is intuitive, and thus, the object system is simple to build. Unfortunately, the closures are also relatively impure from a functional standpoint, and therefore objects built from this system can be problematic. Because of

the nature of the language, use of the `set!` command has issues with both efficiency and referential transparency, and is therefore somewhat unsafe. However, the ease of creating the object system remains a point in favor of the Scheme lexical closures.

Ease of Use

The author found that the Scheme system is relatively easy to use. The concept of lexical closures lies at the heart of Scheme programming and therefore should not be difficult for a Scheme programmer to grasp. Programmers who are inexperienced with Scheme should not find the model difficult to understand since the lexical closures rely on lexical scoping, a common concept in modern programming. The syntax of Scheme is simple to learn, and lexical closures are no exception. With a simple system that is somewhat comparable to the general programming concept of namespaces and a syntax that is easily understandable, Scheme is one of the most accessible systems for beginners and experienced programmers alike.

Speed

The speed aspect of the Scheme environment was the least impressive. We ran the tests on the SCM and MzScheme scheme interpreters. Tables 5.3 and 5.4 show the results of our tests. Although the Windows timing mechanism was not as precise as the Unix time function, we are nonetheless able to see that the performance Scheme was good if not as good as that of J.

5.1.3 Haskell Evaluation

In this section we evaluate the state emulation implementation offered in the Haskell language by the criteria we've seen above. We begin with the usefulness in terms of object-

	MzScheme	SCM
Minimum Allocation	2.17×10^6	2.0×10^4
Maximum Allocation	5.08×10^6	1.0×10^5
Average Allocation	4.49×10^6	6.08×10^4
Minimum Manipulation	1.48×10^6	2.5×10^4
Maximum Manipulation	2.33×10^6	1.0×10^5
Average Manipulation	2.28×10^6	7.25×10^4

Table 5.3: Scheme UNIX tests (in operations/sec)

	MzScheme
Minimum Allocation	8.42×10^5
Maximum Allocation	9.28×10^5
Average Allocation	8.18×10^5
Minimum Manipulation	8.42×10^5
Maximum Manipulation	2.00×10^6
Average Manipulation	1.83×10^6

Table 5.4: Scheme Windows tests (in operations/sec)

oriented programming.

Object-Oriented Programming

One of the drawbacks of the Haskell state emulation method is its usefulness in terms of object orientation. Some attempts have been made to introduce an object-oriented approach to Haskell, such as O'Haskell or Haskell++; these are complex and experimental. Because of the encapsulated nature of the State Monad, object orientation is effectively rendered difficult to achieve using this method. Since the mechanism of the State Monad does not allow for any interaction during the state processing between the state object and the rest of the environment, any sort of object system must be inaccessible for the duration of its existence. This is a poor way to run such a computation and because of this the Haskell State Monad proves a poor tool for implementing object-orientation.

Ease of Use

The ease of use of the State Monad is slightly more difficult to gauge than the object-orientation application. The language of Haskell is understandable and readable, and thus the State Monad syntax is not problematic. However, the concepts required for an in-depth understanding of this system often involve highly theoretical math; this is unsurprising, since the concept for the monad evolved out of category theory. This is balanced, however, by a thriving online community of Haskell programmers and substantial documentation that addresses the State Monad and the other complex features of this language. Though the theory is often difficult to navigate, a plethora of tutorials are available online, some assuming knowledge of mathematical constructs, some explaining the mathematics, and some skirting the issue altogether. Though simplicity is not a feature of this system, the availability of documentation and support often make up for the challenges of learning it.

Speed

The speed tests were run within the interactive environment rather than through a compiled executable in order to fairly compare with the other languages. They were run in Hugs and GHCi, and tables 5.5 and 5.6 show the results of the tests. In the second example, the inaccuracy of the Windows timing mechanism may contribute to the extremely regular data, but in both examples it is clear that Haskell is slower than the other two languages. The reasons for the slowness are likely caused by the intepreted nature of our tests as well as perhaps the implementation that was used (Control.State.Monad was used in the interactive GHC compiler). Thus Haskell was the slowest of the three languages.

	Hugs	GHCi
Minimum Allocation	3.85×10^4	50
Maximum Allocation	4.24×10^5	83
Average Allocation	4.17×10^4	58
Minimum Manipulation	3.57×10^4	50
Maximum Manipulation	4.1×10^5	62
Average Manipulation	3.92×10^4	58

Table 5.5: Haskell UNIX tests (in operations/sec)

	Hugs	GHCi
Minimum Allocation	3.37×10^3	64
Maximum Allocation	3.37×10^3	64
Average Allocation	3.37×10^3	64
Minimum Manipulation	4.0×10^3	64
Maximum Manipulation	4.0×10^3	64
Average Manipulation	4.0×10^3	64

Table 5.6: Haskell Windows tests (in operations/sec)

5.2 Comparative Evaluation

5.2.1 Model Evaluation

The approaches examined here differ widely and strong arguments can be made for any of them. As we evaluate the different methods of emulating state within these languages, we must keep in mind that the examples given were minor and were primarily focused on object-orientation. This seemed to be one of the best ways to judge the capabilities of each language within space limitations, but was slightly biased against Haskell and towards Scheme and J. Nonetheless, each language had certain advantages and disadvantages over the others. For example, the author found that Scheme notation was the simplest to understand from a syntactical standpoint. In fact, much like other facets of the language, the Scheme lexical closures were immediately intuitive. Of course, this simplicity comes with a price. The `set!` operation is not purely functional. In fact, it is an imperative command nestled

inside a functional language. The very presence of the command indicates that it is not technically emulating state, but actually creating it. However, due to the functional nature of Scheme, it is considered an emulation, since most Scheme constructs are put in place to facilitate functional programming. Of course, assigning new values to already existing names causes some problems. For example, referential transparency is broken; that is, if one were to replace the function with the value it returns in every instance where it is called, the program output would be different. These side effects can increase the difficultly of reasoning about the program. The J code, on the other hand, neatly sidesteps much of this while still maintaining an intuitive interface. Like Scheme, J allows the user to assign new values to existing names; however, this is done only to names scoped within a locale. Using this method, names bound within functions do not change, so many negative side effects are prevented. Unfortunately, this comes at the cost of some ease of use. To manipulate the Scheme interface, the programmer need only be familiar with static scoping and state change, which are common concepts. However, while not unknown, the idea of locales used in J is more obscure. In addition, the special "z locale" and the unique way in which these locales are used are more complex than the idea of lexical closures. Both of these are far easier to understand and implement than the Haskell method. The prototype for the special Haskell monads was the monoid of abstract algreba, and this innately presents some challenges to less mathematically knowledgeable programmers. Although an understanding of category theory is not required to effectively use Haskell monads, the interface is still a bit difficult without a basic understanding of the mathematical constructs. The difference between a data type and a State data type is subtle, and the three combinators might be confusing. Not only is the learning curve of Haskell steep, but the capabilities it provides for state emulation are not as useful on a small scale as those in J and Scheme. However, the primary advantage that displaces all of these disadvantages is that Haskell remains purely

functional, with no side effects. Despite the complicated logic surrounding state change in the other two languages, an object still changes state at some point; however, in Haskell, no state change is ever actually evoked. The program behaves as though state has changed, and it certainly appears to be imperative code, but the structure of the program is still referentially transparent. So Haskell has captured the best of two paradigms; it allows the programmer the ease of writing imperative code alongside the ease of reasoning about functional code. However, the programmer must become well versed in Haskell for this to be achieved.

5.2.2 Object-Orientation

J performs best of the three languages reviewed here, simply because of the safety and power it offers. Scheme offers flexibility and power, but it contradicts the functional paradigm in doing so. Scheme, being the only impure language reviewed in this paper, obviously is less safe and more prone to breaking referential transparency. Haskell has the opposite problem, offering encapsulation and safety but preventing the user from creating any useful object-oriented state objects. J is an excellent middle ground between the two, with a sufficiently powerful system that allows encapsulation and retains pure functionality.

5.2.3 Ease of Use

With simple syntax and basic concepts, Scheme outperforms both J and Haskell. In terms of syntax, both Scheme and Haskell are more accessible than J; Scheme has less vocabulary than Haskell, but Haskell has more similarities with modern programming languages (both functional and nonfunctional). Conceptually, Haskell is the most difficult, with ties to advanced mathematics that may elude many programmers. In contast, J is complex to allow some power but has simple concepts, and Scheme is the easiest to understand conceptually

	Hugs	GHCi	MzScheme	SCM	J Console
Minimum Allocation	3.85×10^4	50	2.17×10^6	2.0×10^4	2.97×10^8
Maximum Allocation	4.24×10^5	83	5.08×10^6	1.0×10^5	6.02×10^8
Average Allocation	4.17×10^4	58	4.49×10^6	6.08×10^4	2.69×10^8
Minimum Manipulation	3.57×10^4	50	1.48×10^6	2.5×10^4	2.69×10^8
Maximum Manipulation	4.1×10^5	62	2.33×10^6	1.0×10^5	6.08×10^8
Average Manipulation	3.92×10^4	58	2.28×10^6	7.25×10^4	5.61×10^8

Table 5.7: Haskell tests (in operations/sec)

because of the simplicity of the concept of closures. For online community and documentation, Haskell and Scheme both have large, thriving communities with substantial literature available for all levels of knowledge and experience; J has a helpful manual, but little community and few alternative tutorials. Overall, the author found Scheme to be easiest to use, with Haskell being the second most difficult and J being a demanding language.

5.2.4 Speed

In terms of speed J was quantitatively the best, followed by Scheme and then Haskell. It might be noted, though, that a plethora of implementations exist of both Scheem and Haskell, and while it is beyond the scope of this paper to provide comprehensive testing of a large number of these, it is unlikely that all perform equally at state manipulation. Nevertheless, J seems to continually outperform both Scheme and Haskell at speed tests. Tables 5.7 and 5.8 demonstrate this comparison visually.

	Hugs	GHCi	MzScheme	J Console
Minimum Allocation	3.37×10^3	64	8.42×10^5	1.89×10^8
Maximum Allocation	3.37×10^3	64	9.28×10^5	6.59×10^8
Average Allocation	3.37×10^3	64	8.18×10^5	5.77×10^8
Minimum Manipulation	4.0×10^3	64	8.42×10^5	1.90×10^8
Maximum Manipulation	4.0×10^3	64	2.00×10^6	6.61×10^8
Average Manipulation	4.0×10^3	64	1.83×10^6	5.58×10^8

Table 5.8: Windows tests (in operations/sec)

Chapter 6

Conclusion

Each of the three language examined has unique features that mark it for certain tasks. Haskell is clearly the language of choice for purists and those concerned with referential transparency. J is efficient and powerful, but is more likely to break referential transparency with top-level reassignments. Scheme is well equipped for scripting and highly impure functional programming where ease of programming is paramount to all other considerations. The paradigm in which stateful computation is pursued should be dependent upon the goals of the computation and the preference of the programmer.

Bibliography

[1] Alan Bawden and Jonathan Rees. Syntactic closures. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 86–95, New York, NY, USA, 1988. ACM Press.

[2] Roger Hui and Kenneth Iverson. *J Dictionary.* Iverson Software, Toronto, Ontario, 1998.

[3] Eric Iverson. *J Primer.* Iverson Software, Toronto, Ontario, 1998.

[4] John Launchbury and Simon L. Peyton Jones. State in haskell. *Lisp Symb. Comput.*, 8(4):293–341, 1995.

[5] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995.

[6] Julie Sussman, Harold Abelson, and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT, Cambridge, 1985.

[7] Philip Wadler. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, New York, NY, USA, 1990. ACM Press.

Appendix A

Speed Test Information

A.1 System Specifications

All of the code was tested on a Linux machine with 3.0 Ghz Pentium Processor running Fedora Core 4 with 1G RAM. We ran these tests multiple times to approximate the efficiency for an average user.

A.2 J Tests

We ran our J tests with the following J code:

```
make_z_ =: 0 !: 0 @ <

s =: 0

semaphore_type =: 'semaphore'

semaphore_get =: monad def 's'

semaphore_inc =: monad def 's =: >:s'
```

```
semaphore_dec =: monad def 's =: <:s'
```

We ran the tests using the native J time function, `(6!:2)`, and averaged the result of our tests. The testing was done in J version 5.04.

A.3 Scheme Tests

We ran the Scheme tests with the following Scheme code:

```
(define semaphore
  (let ((val 0))
    (lambda (cmd)
      (cond
        ((equal? cmd "get") val)
        ((equal? cmd "up")
         (set! val (+ val 1)))
        ((equal? cmd "down")
         (set! val (- val 1)))
        (else "Invalid Command")))))

(define semx semaphore)
```

These test were performed using the native Scheme time function, `(get-internal-run-time)` and divided our results by the constant `internal-time-units-per-second`. The testing environment was SCM version 5.1.

A.4 Haskell Tests

We ran the Haskell tests with the following Haskell code:

```
import Monad
import System
import IO
import Control.Monad.State
```

```
semaUp :: State Int ()
semaUp = do {x <- get; put (x+1)}
semaDn :: State Int ()
semaDn = do {x <- get; put (x-1)}
semaMk :: Int -> State Int ()
semaMk n = put n
```

These tests were run with the `getClockTime` function in the Haskell module `System.Time`. As mentioned in the thesis, the tests were run in the interactive environment of GHC version 6.4.1 in order to be more comparable to the Scheme and J tests, both of which were on interpreted code.